COMP1730/COMP6730
Programming for Scientists

Testing and Defensive Programming.

Overview

* Testing
* Defensive Programming

Overview of testing

* The purpose of testing is to detect **bugs**. There are many different types of testing - load testing, integration testing, user experience testing, etc.
* Different software systems have different testing requirements, based on:
  - Consequences of failure
  - Complexity of software
  - Frequency of use
  - Hardware and user interactions
* Even for critical, commercially developed software, testing gives no guarantees - e.g. Boeing 737 Max crashes.

Unit-Testing

* We are concerned with *unit-testing* or functional testing.
* Usually done at the function (or method level).
* Done by calling a function with specified parameters (inputs) and checking that the return value (output) is as expected, called **test cases**.

Good test cases

* Satisfy the assumptions.
* Simple (enough that correctness of the value can be determined “by hand”).
* Cover the space of inputs *and* outputs.
* Cover branches in the code.
* We usually want to focus on *edge-cases*:
  - Integers: 0, 1, -1, 2, ...
  - `float`: very small (1e-308) or big (1e308)
  - Sequences: empty (`''`, `[]`), length one.
  - Any value that requires special treatment in the code.

The `assert` Statement

* Basic usage:

```
assert boolean_expression
assert boolean_expression, "message"
```

* If the expression is `True` execution continues.
* If the expression is `False` an `AssertionError` is raised, execution stops and the message is printed.
* Can be used to intentially cause a run-time error if assumptions are violated.

Example from homework 2

```
def test_combinations():
    """This function runs a number of tests of combinations function.
    If it works ok, you will see the output ("all tests passed") at
    the end when you call this function; if some test fails, there will
    be an error message."""
    # simple test cases:
    assert combinations(5, 2) == 10
    assert type(combinations(5, 2)) is int
    assert combinations(5, 3) == 10
    # number of possible 5-card hands from a deck of 52 cards:
    assert combinations(52, 5) == 2598960
    # some edge cases:
    assert combinations(0, 0) == 1
    assert combinations(1, 0) == 1
    assert combinations(1, 1) == 1
    assert combinations(100, 0) == 1
    assert combinations(100, 100) == 1
    print("all tests passed")
```

Other Testing Considerations

* Floating point precision
* Random numbers (use a *seed* to get reproducable results).
* User input (isolate the user input to a function and simulate input).
* Only use your code to generate tests for refactoring purposes, not for testing correctness.
* **Testing only guarantees your code works for the test cases!**

Defensive programming

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

* Write code that is easy to read and well documented.
  - If it's hard to understand, it's harder to debug.

Code Quality Matters!

* A function that is hard to read is hard to debug.

```
def AbC(ABc):
    ABC = len(ABc)
    ABc = ABc[ABC-1:-ABC-1:-1]
    if ABC == 0:
        return 0
    abC = AbC(ABc[-ABC:ABC-1:])
    if ABc[-ABC] < 0:
        abC += ABc[len(ABc)-ABC]
    return abC
```

* A small function that only does one thing is easier to test than a large function that does many things.

Pre and Post Conditions

* `assert` statements allow us to ensure that only appropriate parameters are passed as arguments to functions. Example:

```
assert type(param_a) == int and param_a > 0
```

Bad practice (delayed error):

```
def sum_of_squares(n):
    if n < 0:
        return "error:\ n is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6
m = ...
k = ...
a = sum_of_squares(m)
b = sum_of_squares(m - k)
c = sum_of_squares(k)
if a - b != c:
    print(a, b, c)
```

Good practice (immediate error):

```
def sum_of_squares(n):
    assert n >= 0, str(n) + " is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6
```

Explicit vs Implicit

* Make things explicit if they are unclear or could be confusing. Even if they are working as intended.
* `return None` is better than no return statement.
* `- (2 ** 2)` instead of `- 2 ** 2`.
* `(a and b) or c` instead of `a and b or c`.
* `dict()` instead of `{ }`.

Avoid Language Tricks

* Don’t make use of language quirks in your code.
* Example: operator chaining.

```
>>> 1 == 2
False
>>> False is not True
True
>>> 1 == 2 is not True
???
```

Design test cases for the following function, regardless of how it is implemented:

```
def hamming_distance(x, y):
    """Compute the Hamming distance between two sequences x and y
    of the same length, defined as the number of corresponding
    i–th elements in x and y that are different"""
```

For example:

* hamming_distance([1, 4, 7, 9, 5], [1, 3, 7, 2, 5]) is 2 (because they differ at index 1 and 3, but are equal at index 0, 2 and 4)
* hamming_distance(”ACCGAT”, ”CACGGA”) is 4 (because they differ at index 0, 1, 4 and 5).

Test design: considerations

* Different sequence types: string, list, tuple.
* Edge-cases: input empty string, input empty list, output zero distance, output maximum distance.
* Mixed data: e.g., `[1, 2, "a", "b", "c"]`
* List of lists: e.g., `[[1,2,3], 4, "a", []]`

```
#easy cases
x=[1, 4, 7, 9, 5], y=[ 1, 3, 7, 2, 5], out=2
x="ACCGAT", y="CACGGA", out=4
x=(1,2), y=(1,2), out=0
x=[3, 3, 2, 5, 1, 7, 6], y=[3, 3, 5, 2, 1, 7, 4], out=3

#edge cases
x=[], y=[], out=0
x='', y='', out=0
x=[1, 5, 2], y=[1, 5, 2], out=0
x=[3, 6, 2, 5], y=[6, 3, 5, 2], out=4
x="abcdefg", y="ABCDEFG", out=7

#difficult case: mixed data
x=[1,2,3,"a","b","c"], y=[1,2,3,"a","b","c"], out=0
x=[1, "a", 13, 20, "b", 0], y=[13, "a", 1, 20, "c", 0], out=3
x=[[1,2,3], 4, "x", [], []], y=[[1,2,3], 6, "x", ["a","b"],[]], out=2
```

Take home messages

* Bugs are unavoidable and testing is therefore essential for software development.
* Unit-testing: design ”good” test cases that cover space of inputs and outputs and edge-cases and difficult cases.
* Try to write good quality code: they are easier to debug.